University of North Carolina at Charlotte

8-Puzzle Program Using A* Search

PROJECT REPORT

ITCS 6150
Intelligent Systems

SUBMITTED TO:
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:
Jacob Gulan

I. PROBLEM FORMULATION

1. 8-Puzzle Problem

The 8-puzzle problem is constituted by a three-by-three grid of blocks labeled 1 through 8 with one grid space being empty. The goal of the puzzle is to take the puzzle from its original state to the desired goal game by sliding the blocks into the empty spot until the goal state is achieved. Figure 1 shows a sequence of legal moves that correctly transition the puzzle from the initial state to the goal state.

Figure 1. Sequence of moves from initial state to goal state

2. A* Search

The 8-puzzle problem can be solved through the implementation of an A* search algorithm. A* search is a type of best-first search algorithm that minimizes the total estimated solution cost. Utilizing A* search for the 8-puzzle problem ensures that the chosen path from the initial state to the goal state is the shortest path possible.

A* search is capable of determining the most optimal solution by calculating the values of g(n), h(n), and f(n). The estimated cost of the path from the node to the goal is denoted by f(n). The path cost from the initial state to the current state (n) is denoted by g(n). Finally, h(n) denotes the heuristic value which estimates the cost from the current state (n) to the goal state.

To calculate g(n), all that is required is counting the number of steps taken from the initial state to the current state. For this project, two heuristic types were implemented: manhattan distance and misplaced tiles. The former calculates h(n) by taking the sum of the horizontal and vertical distance from the current state to the goal state. The latter calculates h(n) by counting the number of misplaced tiles in the current state compared

to the goal state. The cost of the shortest path, f(n), is calculated by adding g(n) and h(n). These cost values allow the A* search algorithm to find the optimal path.

3. Solution Path

This 8-puzzle program accepts user input to determine the initial state and goal state of the puzzle. If a solution is not possible then the program will return that it is not solvable. Otherwise, the program will output a solution path from the initial state to the goal state. The program will also output the number of generated nodes from the frontier list and expanded nodes from the explored list.

II. PROGRAM STRUCTURE

1. Classes

Class	Class description
Node	The Node class is responsible for holding and calculating the values of g(n), h(n), f(n), as well as the current state of the puzzle. It also collects the future possible moves to be added to the frontier list by retrieving the neighboring nodes of the empty block.
AStarSearch	The AStarSearch class is responsible for executing the bulk of the code. The class maintains the frontier and explored list while also receiving user input in regard to the initial state, goal state, and heuristic type. It also prints out of the solution path and the amount of generated and expanded nodes.

2. Local Variables

Node Class

Variable	Variable type
value	Numpy array of puzzle's state
gn	Integer holding path cost
fn	Float holding estimated cost of the cheapest solution through n
hn	Float holding estimated cost of the cheapest path from the state at node n to a goal state

AStarSearch Class

Variable	**Variable type**
openList	Numpy array of the frontier list
closedList	Numpy array of the explored list

3. Functions and Procedures

Node Class

Function/Procedure	**Description**
__init__(self, v, g, f)	The initializer of the AStarSearch class creates an object of the class and instantiates the value, gn, fn, and hn. This function creates the nodes that will be added to the frontier list to determine the optimal path. Params: v - Numpy array *(the state of the puzzle)* g - Integer *(the path cost, g(n))* f - Float *(the estimated cost of the cheapest solution through n, f(n))*
updateHeuristic(self, htype, goal)	This function updates the heuristic value of hn and the value of fn. This function ensures hn and fn get calculated and assigned the proper values. Params: htype - Integer *(value to determine if manhattan or misplaced tiles heuristic)* goal - Numpy array *(goal state)*
getNeighbors(self)	This function finds the location of the empty block (the zero node) and collects its neighboring nodes. The function ensures everything stays within the bounds of the puzzle. Once a neighbor is found it is then added to the list of neighbors and eventually returned and added to the frontier list. Return: List *(neighboring nodes of the empty block)*
isInBounds(self, x, y)	This is a helper function that assists the

	getNeighbors() function in determining if a possible neighbor is within the puzzle. Params: x - Integer *(the x-value of the neighbor)* y - Integer *(the y-value of the neighbor)* Return: Boolean *(true if neighbor is in bounds and false if outside of bounds)*
manhattan(self, goal)	The manhattan function determines the heuristic cost h(n) of the current state through the goal state using the manhattan distance heuristic. Manhattan distance is calculated by taking the horizontal and vertical distance between the current state and the goal state. Params: goal - Numpy array *(the goal state)* Return: cost - Float *(the heuristic cost of manhattan distance)*
misplaced(self, goal)	The misplaced function determines the heuristic cost h(n) of the current state through the goal state using the misplaced tiles heuristic. The misplaced tiles cost is calculated by determining how many nodes are not in their correct position. Params: goal - Numpy array *(the goal state)* Return: cost - Float *(the heuristic cost of misplaced tiles)*

AStarSearch Class

Function/Procedure	**Description**
__init__(self)	The initializer of the AStarSearch class creates an object of the class and instantiates the frontier and explored lists denoted as openList and closedList respectively.

main(self)	This function is responsible for executing and carrying out the A* search algorithm as well as printing the results. The user inputs the initial state, goal state, and heuristic value in this function. If the inputted values are valid then the main() function will either output that the puzzle is not solvable or will output the path of the solution as well as the amount of generated and expanded nodes while searching.

III. Input and Output Cases

1. Case 1

Manhattan

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 1 3 4 8 0 5 7 2 6
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 8 0 4 7 6 5
Input 0 for Manhattan and 1 for Misplaced Tiles: 0

1 3 4
8 0 5
7 2 6

1 3 4
8 2 5
7 0 6

1 3 4
8 0 5
7 2 6

1 3 4
8 2 5
7 6 0

1 3 4
8 2 0
7 6 5

1 3 0
8 2 4
7 6 5

1 0 3
8 2 4
```

```
7 6 5

1 2 3
8 0 4
7 6 5


Generated Nodes:  15
Expanded Nodes:  7
```

Misplaced Tiles

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 1 3 4 8 0 5 7 2 6
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 8 0 4 7 6 5
Input 0 for Manhattan and 1 for Misplaced Tiles: 1

1 3 4
8 0 5
7 2 6

1 0 4
8 3 5
7 2 6

1 3 4
8 2 5
7 0 6

1 3 4
8 5 0
7 2 6

1 3 4
8 0 5
7 2 6

1 3 4
8 0 5
7 2 6

1 3 4
8 2 5
7 6 0

1 3 0
8 5 4
7 2 6
```

```
1 3 4
8 0 5
7 2 6

1 3 4
8 2 0
7 6 5

1 0 3
8 5 4
7 2 6

1 3 0
8 2 4
7 6 5

1 3 4
8 0 2
7 6 5

1 5 3
8 0 4
7 2 6

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5


Generated Nodes:  34
Expanded Nodes:  15
```

2. Case 2

Manhattan

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 1 3 4 8 6 2 0 7 5
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 8 0 4 7 6 5
Input 0 for Manhattan and 1 for Misplaced Tiles: 0

1 3 4
8 6 2
0 7 5
```

```
1 3 4
8 6 2
7 0 5

1 3 4
8 0 2
7 6 5

1 3 4
8 2 0
7 6 5

1 3 0
8 2 4
7 6 5

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5


Generated Nodes:  12
Expanded Nodes:  6
```

Misplaced Tiles

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 1 3 4 8 6 2 0 7 5
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 8 0 4 7 6 5
Input 0 for Manhattan and 1 for Misplaced Tiles: 1

1 3 4
8 6 2
0 7 5

1 3 4
8 6 2
7 0 5

1 3 4
8 0 2
7 6 5
```

```
1 0 4
8 3 2
7 6 5

1 3 4
8 2 0
7 6 5

1 3 4
8 0 2
7 6 5

1 3 0
8 2 4
7 6 5

1 3 4
8 0 2
7 6 5

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5


Generated Nodes:  20
Expanded Nodes:  9
```

3. Case 3

Manhattan

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 7 4 5 6 8 0
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 8 6 4 7 5 0
Input 0 for Manhattan and 1 for Misplaced Tiles: 0

1 2 3
7 4 5
6 8 0

1 2 3
7 4 0
6 8 5
```

1 2 3
7 4 5
6 8 0

1 2 3
7 0 4
6 8 5

1 2 3
7 8 4
6 0 5

1 2 3
7 8 4
6 5 0

1 2 3
7 8 4
0 6 5

1 2 3
7 4 5
6 0 8

1 2 3
7 4 5
6 8 0

1 2 3
7 4 0
6 8 5

1 2 3
7 4 0
6 8 5

1 2 3
7 8 0
6 5 4

1 2 3
7 8 4
6 5 0

1 2 3
7 8 4
6 0 5

```
1 2 3
7 8 4
6 5 0

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

1 2 3
8 6 4
7 0 5

1 2 3
8 6 4
7 5 0


Generated Nodes:  32
Expanded Nodes:  18
```

Misplaced Tiles

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 7 4 5 6 8 0
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 8 6 4 7 5 0
Input 0 for Manhattan and 1 for Misplaced Tiles: 1

1 2 3
7 4 5
6 8 0

1 2 3
7 4 0
6 8 5

1 2 3
7 4 5
6 0 8

1 2 3
7 4 5
6 8 0

1 2 3
```

7 0 4
6 8 5

1 2 3
7 4 5
6 8 0

1 2 3
7 0 5
6 4 8

1 2 3
7 4 5
0 6 8

1 2 3
7 8 4
6 0 5

1 2 3
7 8 4
6 5 0

1 2 3
0 7 4
6 8 5

1 2 3
0 4 5
7 6 8

1 2 0
7 4 3
6 8 5

1 2 3
7 4 0
6 8 5

1 2 3
7 4 5
6 0 8

1 0 3
7 2 4
6 8 5

1 2 3

```
7 4 0
6 8 5

1 2 3
7 4 0
6 8 5

1 2 3
7 4 5
6 0 8

1 2 3
7 4 5
6 0 8

1 2 3
0 7 5
6 4 8

1 2 3
7 5 0
6 4 8

1 2 3
7 4 5
6 0 8

1 2 3
7 0 4
6 8 5

1 2 3
7 8 4
0 6 5

1 2 3
6 7 4
0 8 5

1 2 3
7 0 4
6 8 5

1 2 3
4 0 5
7 6 8

1 2 3
```

7 4 0
6 8 5

1 2 3
7 4 5
6 8 0

1 2 3
7 0 4
6 8 5

1 2 3
7 4 5
6 8 0

1 2 3
7 0 4
6 8 5

1 2 3
7 4 5
6 8 0

1 2 3
7 0 4
6 8 5

1 2 3
7 4 5
6 8 0

1 2 3
7 0 4
6 8 5

1 2 3
7 4 5
6 8 0

1 2 3
7 4 5
6 8 0

1 2 3
7 5 8
6 4 0

1 2 3

```
7 4 5
6 8 0

1 2 3
0 8 4
7 6 5

1 2 3
4 6 5
7 0 8

1 2 3
7 4 5
6 8 0

1 2 3
7 0 4
6 8 5

1 2 3
8 0 4
7 6 5

1 2 3
4 6 5
7 8 0

1 2 3
8 6 4
7 0 5

1 2 3
8 6 4
7 5 0


Generated Nodes:  90
Expanded Nodes:  48
```

4. Case 4

Manhattan

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 0 1 2 3 4 5 6 7 8
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 4 5 6 7 8 0
Input 0 for Manhattan and 1 for Misplaced Tiles: 0

Not Solvable
```

Misplaced Tiles

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 0 1 2 3 4 5 6 7 8
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 4 5 6 7 8 0
Input 0 for Manhattan and 1 for Misplaced Tiles: 1

Not Solvable
```

5. Case 5

Manhattan

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 2 8 1 3 4 6 7 5 0
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 3 2 1 8 0 4 7 5 6
Input 0 for Manhattan and 1 for Misplaced Tiles: 0

2 8 1
3 4 6
7 5 0

2 8 1
3 4 0
7 5 6

2 8 1
3 0 4
7 5 6

2 0 1
3 8 4
7 5 6

2 8 1
3 0 4
7 5 6

0 2 1
3 8 4
7 5 6

3 2 1
0 8 4
7 5 6

3 2 1
8 0 4
7 5 6
```

```
Generated Nodes:  15
Expanded Nodes:  7
```

Misplaced Tiles

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 2 8 1 3 4 6 7 5 0
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 3 2 1 8 0 4 7 5 6
Input 0 for Manhattan and 1 for Misplaced Tiles: 1

2 8 1
3 4 6
7 5 0

2 8 1
3 4 0
7 5 6

2 8 1
3 0 4
7 5 6

2 0 1
3 8 4
7 5 6

2 8 1
0 3 4
7 5 6

2 8 1
3 0 4
7 5 6

0 2 1
3 8 4
7 5 6

2 8 1
3 0 4
7 5 6

3 2 1
0 8 4
7 5 6
```

```
3 2 1
8 0 4
7 5 6


Generated Nodes:  20
Expanded Nodes:  9
```

6. Case 6

Manhattan

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 7 2 4 5 0 6 8 3 1
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 4 5 6 7 8 0
Input 0 for Manhattan and 1 for Misplaced Tiles: 0

Not Solvable
```

Misplaced Tiles

```
Input initial state in a similar format of '0 1 2 3 4 5 6 7 8': 7 2 4 5 0 6 8 3 1
Input goal state in a similar format of '0 1 2 3 4 5 6 7 8': 1 2 3 4 5 6 7 8 0
Input 0 for Manhattan and 1 for Misplaced Tiles: 1

Not Solvable
```

IV. RESULTS AND CONCLUSION

Case	**Manhattan**	**Misplaced Tiles**
Case 1	Generated Nodes: 15 Expanded Nodes: 7	Generated Nodes: 34 Expanded Nodes: 15
Case 2	Generated Nodes: 12 Expanded Nodes: 6	Generated Nodes: 20 Expanded Nodes: 9
Case 3	Generated Nodes: 32 Expanded Nodes: 18	Generated Nodes: 90 Expanded Nodes: 48
Case 4	Not solvable	Not solvable
Case 5	Generated Nodes: 15 Expanded Nodes: 7	Generated Nodes: 20 Expanded Nodes: 9
Case 6	Not solvable	Not solvable

Figure 2. Case results

Based on the results in figure 2, it's evident that the manhattan heuristic performs better than the misplaced tiles heuristic when it comes to determining the shortest path. In case 1 and case 2, the amount of generated nodes and expanded nodes from the misplaced tiles exceeded more than double that of the Manhattan heuristic.

The A* Search algorithm optimally found the best path for both heuristic values, but it appears the Manhattan heuristic should be favored over the misplaced tiles heuristic to ensure the smallest-path is found.

V. REFERENCES

1. https://www.geeksforgeeks.org/counting-inversions/
2. https://www.d.umn.edu/~jrichar4/8puz.html
3. https://www.cs.ubc.ca/~kevinlb/teaching/cs322%20-%202008-9/Lectures/Search5.pdf
4. https://www.cs.cmu.edu/~eugene/teach/ai00/sample/index.html
5. https://github.com/Jeet0204/8-Puzzle-using-AStar-Algorithm/blob/master/8Puzzle_A_star_search_algorithm.ipynb
6. https://cs.calvin.edu/courses/cs/344/kvlinden/resources/AIMA-3rd-edition.pdf#page=821&zoom=100,0,0
7. https://stackoverflow.com/
8. https://numpy.org/